

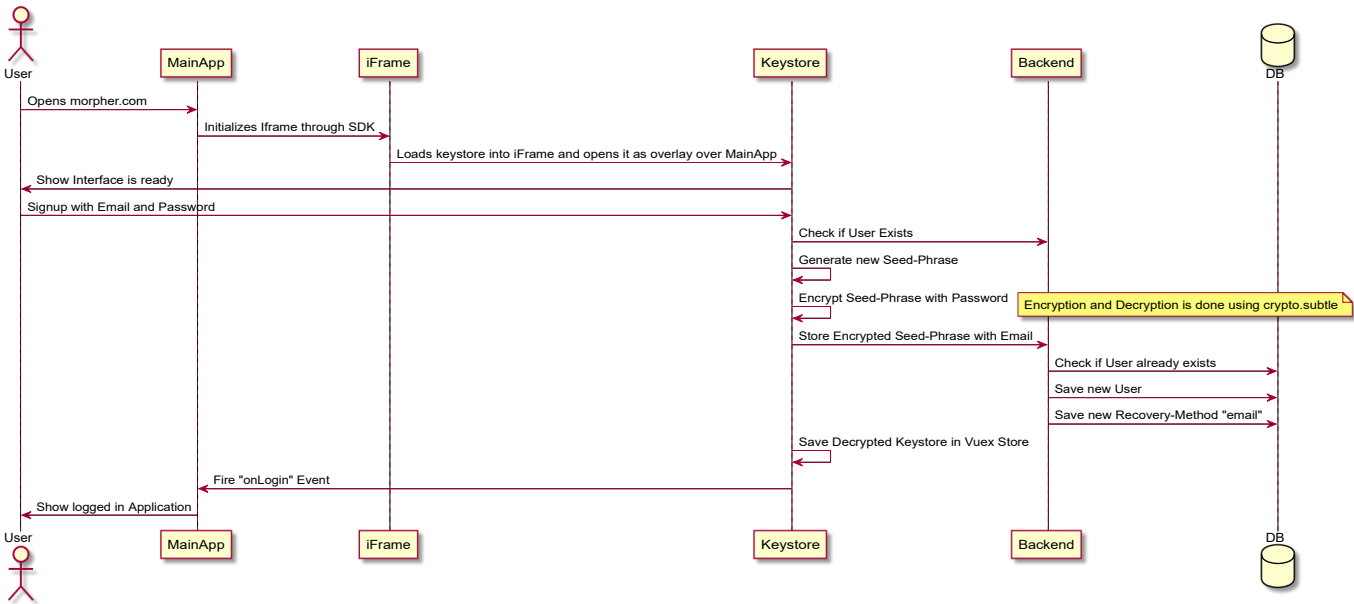
Morpher Wallet

This is the guideline for a security review of the Morpher Wallet

General thoughts for the wallet

Getting started quickly

Iframe Flow

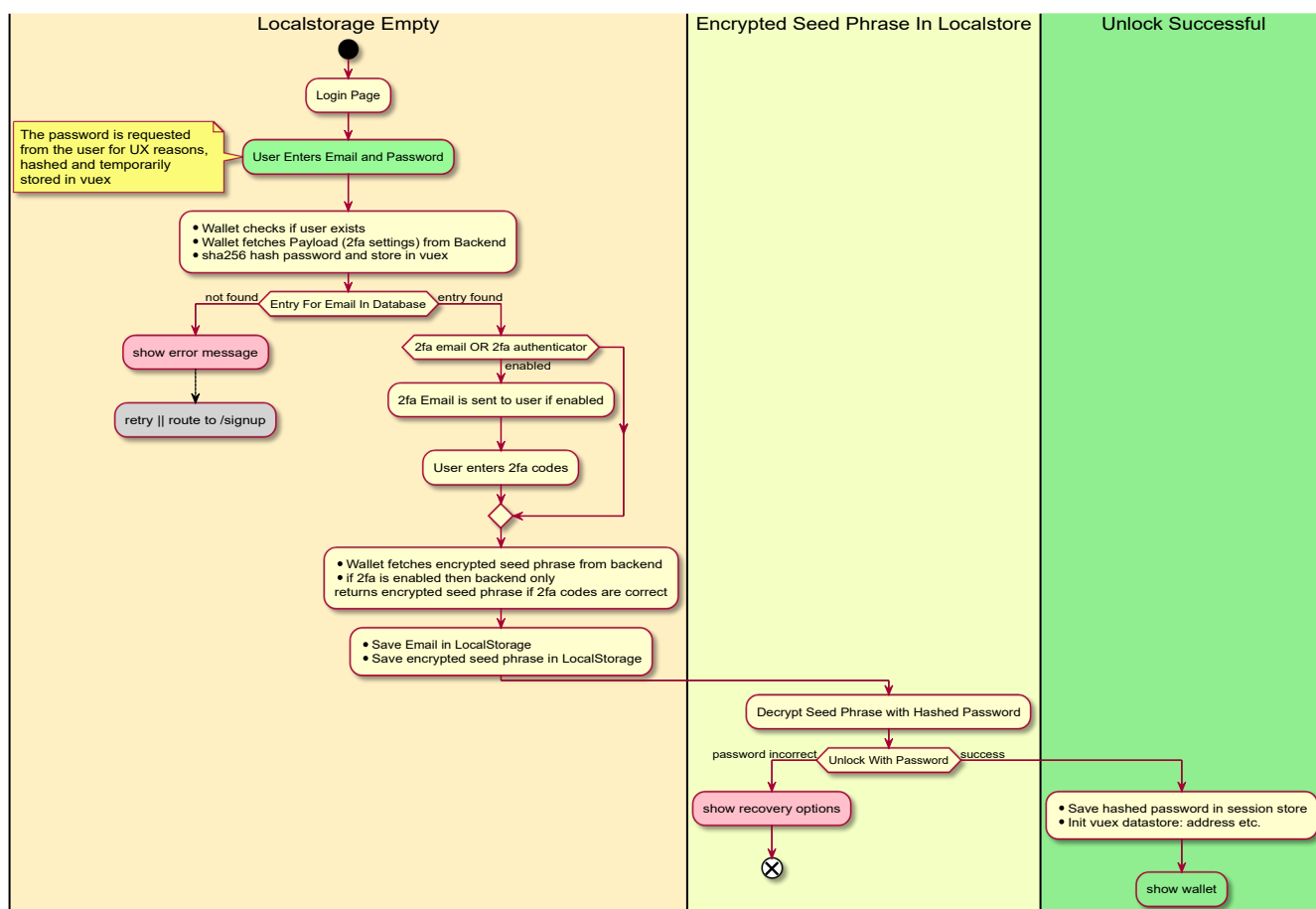


Login Flow for the Wallet

This section describes the login and unlock-Flow for the wallet.

The login and unlock flow are different only by the contents of the localStorage. If the encryptedSeed is already stored in localStorage, then we don't need to fetch it from the backend.

Login Flow



When the user logs into the wallet the first time, then the browsers localStorage and sessionStorage is empty.

1. The user is required to enter Email-Address and Password.

The password is temporarily stored, because of UX reasons. It is used later in the process.

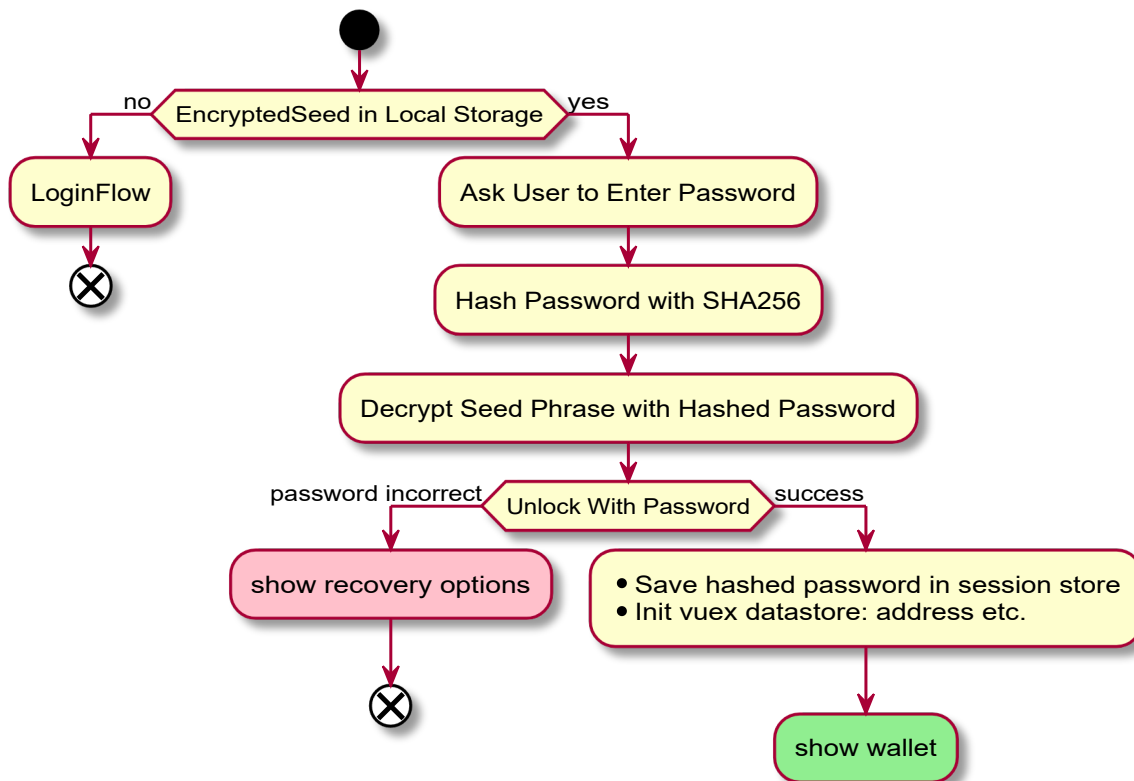
2. The Wallet checks with the backend if the user already exists.

Sidenote: It became "good behavior" to show generic error messages to avoid leaking user-exists information. This simply doesn't make sense, because the backend is a data-store for an encrypted wallet. The actual decryption process is done in-browser.

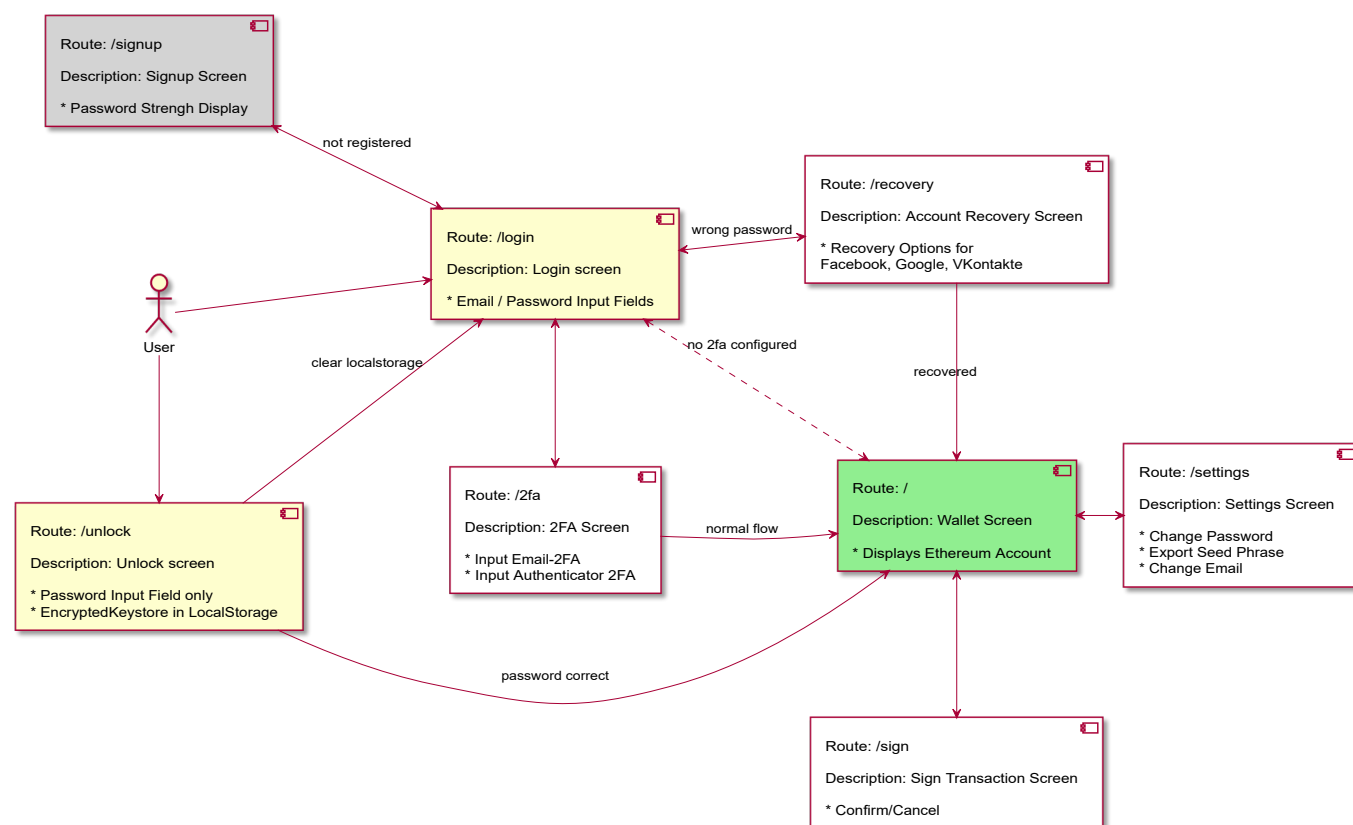
3. Depending on the 2FA settings, the user is then required to enter the 2FA codes. The backend encrypts the wallet with the 2fa codes and only the correct codes + password can decrypt the keystore.

4. If decryption with 2FA and password was successful then the *password-encrypted* keystore (after decrypt with 2fa) is stored in localStorage. This lets the user "unlock" the keystore with the password only in sub-sequent uses of the wallet
5. The wallet is initialized: The keystore is used to generate an account. The account is stored in the JavaScript-Storage-Backend. The frontend shows the Address and UI

Unlock Flow



Description of the Routes



Route	description	Meta-Information	Next Screens Possible
Login: /login	The first screen if the LocalStorage is empty	encryptedKeystoreInStorage == false && keystoreUnlocked == false	Signup, Recovery, 2fa, Wallet
Signup: /signup	If the user wants to create a new Account, then this is the component which takes care of that.	encryptedKeystoreInStorage == false && keystoreUnlocked == false	Login, 2fa
Unlock: /unlock	Let's the user unlock the keystore	encryptedKeystoreInStorage == true && keystoreUnlocked == false	
2FA: /twoFa	If the user is required to enter 2fa codes, then he is redirected here	mail2fa or authenticator2fa = true	Wallet, Login
Wallet: /	If the user is logged in correctly, he will see the wallet address and can switch accounts	keystoreUnlocked = true	Login, Settings, Sign
Settings: /settings	Change Email, Password, Export Seed Phrases, Add Account Recovery	keystoreUnlocked = true	Wallet

Route	description	Meta-Information	Next Screens Possible
Sign Transaction: /signtx	Shows a summary of the transaction and a sign/cancel button	keystoreUnlocked = true	Wallet

Overview

This document describes the external packages used in the Morpher Wallet.

Frontend Technologies

The frontend is split into two parts:

1. The Keystore
2. The iFrame SDK

In general both are written in TypeScript (or ported to TS).

The Keystore is written in VueJs with a Vuex datastore.

The iFrame SDK is written using a native iFrame HTML element and binding parent/child listeners through penpal to the parent/child document. More on the flow below.

List of Tools/Frameworks Used

The following is a non-exhaustive list of important packages used in the Keystore-App with their description:

Web3js Wallet

<https://web3js.readthedocs.io/en/v1.2.0/web3-eth-accounts.html>

The web3.eth.accounts contains functions to generate Ethereum accounts and sign transactions and data.

It's currently marked as "*This package has NOT been audited and might potentially be unsafe. Take precautions to clear memory properly, store the private keys safely, and test transaction receiving and sending functionality properly before using in production!*". We're aware of this fact, no data leakage is indicated at this point.

Web3js is a well maintained open source library and widely in use. Web3.eth.accounts is giving back an object containing scoped functions to sign a transaction. The object is never exposed

```
{
  address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
  privateKey:
    "0x348ce564d427a3311b6536bbcff9390d69395b06ed6c486954e971d960fe8709",
  signTransaction: function(tx){...},
  sign: function(data){...},
  encrypt: function(password){...}
}
```

Web3js can either import private keys or generate them. Since we base the wallet on *seed-phrases*, another library, bip39, is used to generate the seed phrases.

bip39

<https://www.npmjs.com/package/bip39>

JavaScript implementation of Bitcoin BIP39: Mnemonic code for generating deterministic keys

```
const mnemonic = bip39.generateMnemonic()  
// => 'seed sock milk update focus rotate barely fade car face mechanic mercy'
```

This generates a mnemonic using [crypto.getRandomValues](#) in the browser and then fetches words from an english wordlist. The [wordlist](#) is 2048 words long and the entropy generated by default is 128 bits. This gives 16 random bytes using the [randombytes](#) package.

This mnemonic is then piped through `ethereumjs-utils` to generate a private key.

hdkey from ethereumjs-utils

<https://www.npmjs.com/package/ethereumjs-util> is a collection of utility functions for Ethereum.

The Morpher Wallet uses the [hdkey](#) functions from `ethereumjs-wallet`

We use it in `vue/src/utils/keystore.ts` to generate the private keys. Sample code:

```
import { hdkey } from 'ethereumjs-wallet';  
function getPrivateKeyFromMnemonic(mnemonic: string, index: number) {  
  
  const seed = mnemonicToSeedSync(mnemonic);  
  const hdwallet = hdkey.fromMasterSeed(seed);  
  const walletHdPath = "m/44'/60'/0'/0/";  
  
  const wallet = hdwallet.derivePath(walletHdPath + index).getWallet();  
  
  const privateKey = wallet.getPrivateKey().toString("hex");  
  
  return privateKey;  
}
```